# Naming

## Naming concepts

Names are used to refer to entities, such as hosts, users, files and services.

Entities must be referred to by other entities; for example, a host must interact with another one and to do so, it needs to know its reference.

In our case, names are the main way to access some resource.

Entities are usually accessed through an **access point**, which is a special entity characterised by an **address**. An address is just a special case of a name.

The same entity may be accessed through **multiple access points**, both contemporarily or at different times. For this reason, it is **not convenient** to use an address of an access point as the name for its entity. It is better to use **location-independent names**.

We can also make two more distinctions among names:

- **Global vs. local names**: a global name denotes the same entity, no matter where the name is used, while the interpretation of a local name depends on where it is being used
- **Human-friendly vs. machine-friendly names**: a human-friendly name helps a human identify an entity, while a machine-friendly name helps other machines identify an entity (generally, MAC addresses are machine-friendly names)

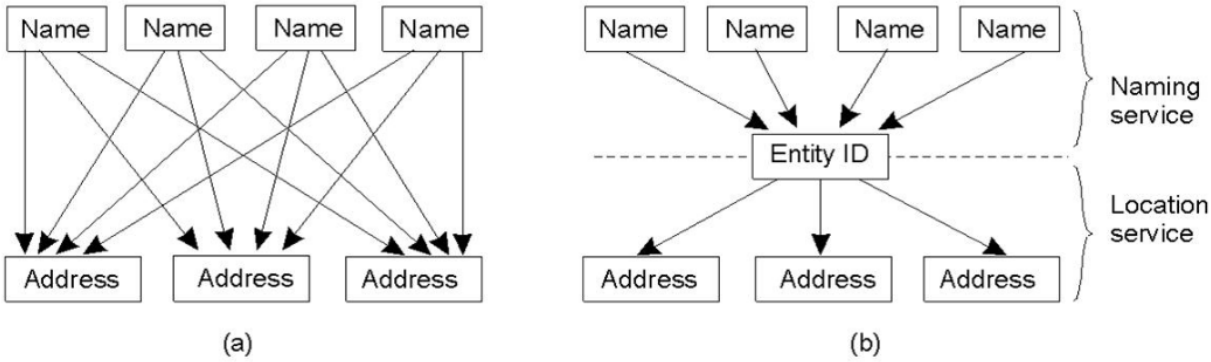
### Identifiers

Resolving a name directly into an address does not work with mobility, so we use **identifiers**.

Identifiers are name such that:

- They never change during the lifetime of the entity
- Each entity has exactly **one** identifier
- An identifier for an entity is never assigned to another entity

Using identifiers enables us to split the problem of mapping a name to an entity and the problem of locating the entity.




## Name resolution

**Name resolution** is the process of obtaining the address of a valid access point of an entity having its name.

Some examples of name resolution are:

- DNS: maps domain names to hosts
- X500 and LDAP: maps a person's name to email, telephone number and more
- RMI registry: maps the name of a Java remote object to its remote reference
- UDDI: maps the description of a Web service to the service metadata needed to access it

The way name resolution is performed depends on the nature of the naming schema employed:

- **Flat** naming
- **Structured** naming
- **Attribute-based** naming

## Flat naming

In a flat naming schema, names are "flat": simple string with **no structure or content**.

Flat naming is designed for **small-scale** (e.g. LAN) environments.

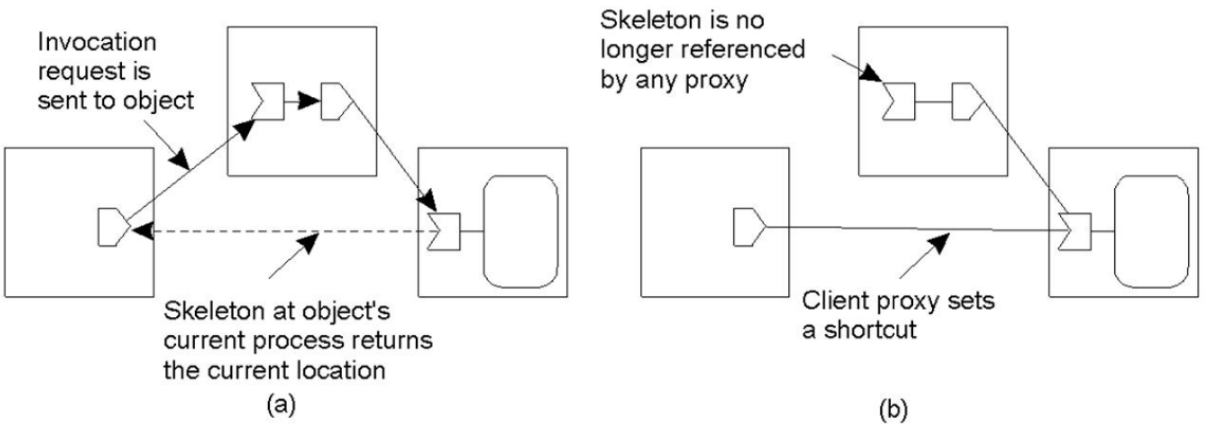
Flat naming has a simple resolution process based on broadcast, multicast or forwarding pointers:

- **Broadcast**: similar to ARP: send a "find" message on a broadcast channel and only the interested machines reply
- **Multicast**: similar to broadcast but it reduces the scope of the search
- **Forwarding pointers** (for mobile nodes): when an entity moves to another location, it leaves a **reference to the next location** at the previous one

Forwarding pointer chains can become very long and broken links cause chains to break. Furthermore, this approach increases network latency.

There must be some clean-up mechanisms to keep the network performant and reliable.

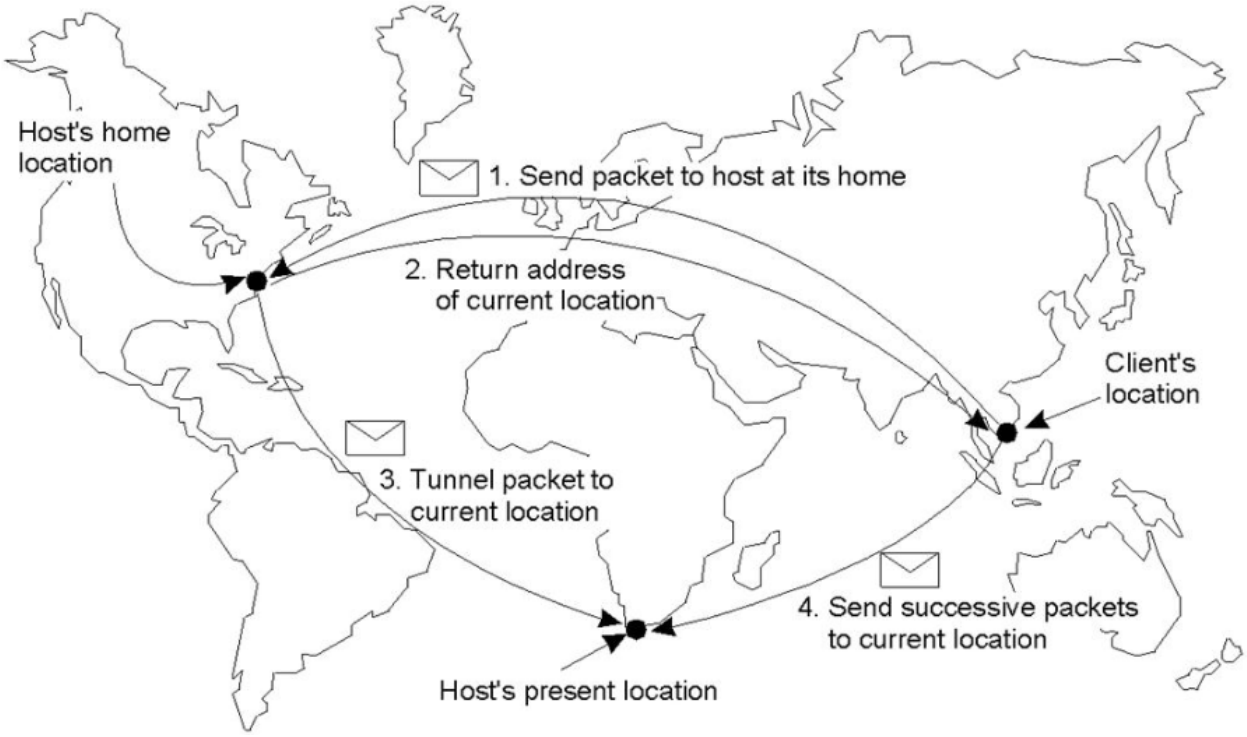
One method to solve this is to use **proxies**, which forward requests to the real object instance. Adjusting proxies enables clean-up: when a skeleton is no longer referenced by a proxy, it can be deleted.




## Home-based approaches

Early cellular network systems used home-based approaches, which are based on forwarding pointers. They relied on one home node that knows the location of the mobile unit.

The home is assumed stable and can be replicated for robustness; furthermore, the original IP of the host is effectively used as an identifier.




Home-based approaches have some problems:

- The extra step towards home **increases latency**
- The home address has to be supported **as long as the entity lives**
- The home address is **fixed**, which becomes an unnecessary burden when the entity permanently moves away
- This method has **poor geographical scalability**

### Distributed Hash Tables (DHT)

In order to identify entities, a hash table can be used. An implementation of DHT consists in nodes organised in a structured overlay network, which can be a ring, a tree, or many others.

Thanks to the DHT, name resolution is quite simple:

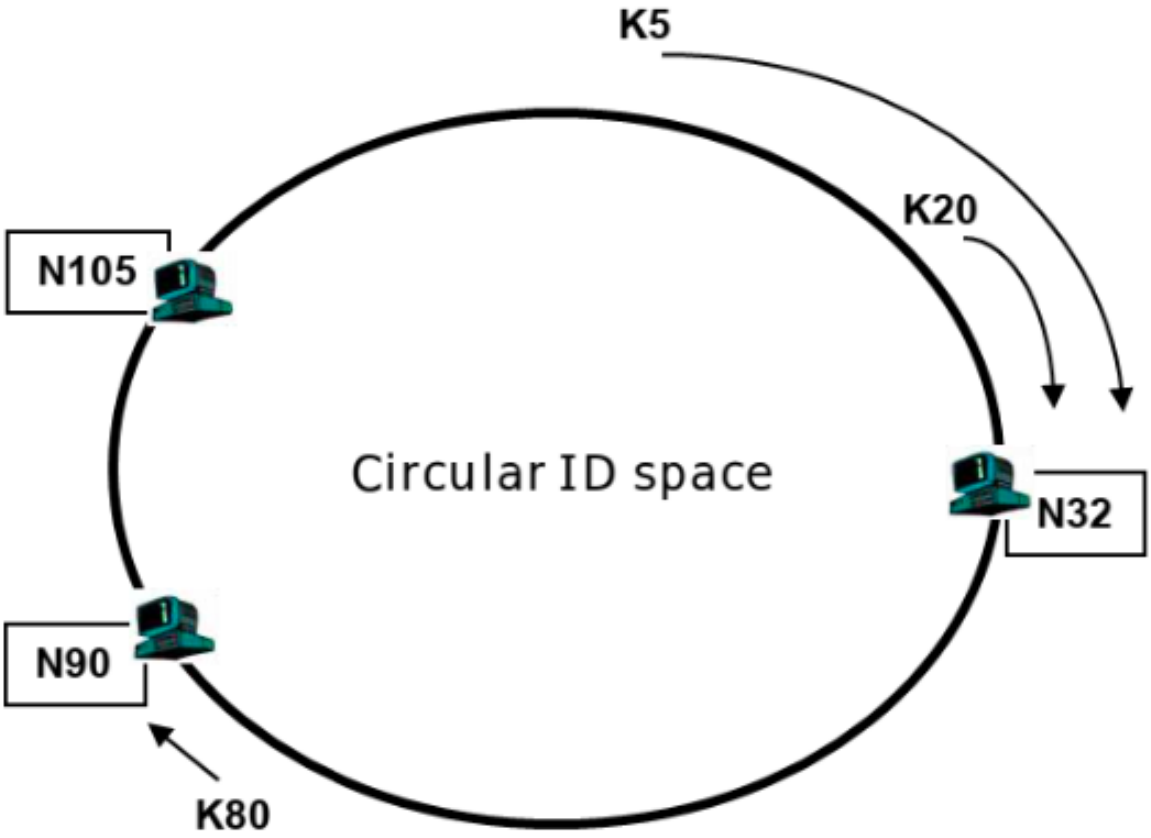
- The entity ID is linked to the name of the entity
- The item to reference is linked to its identifier or address

#### Example: Chord

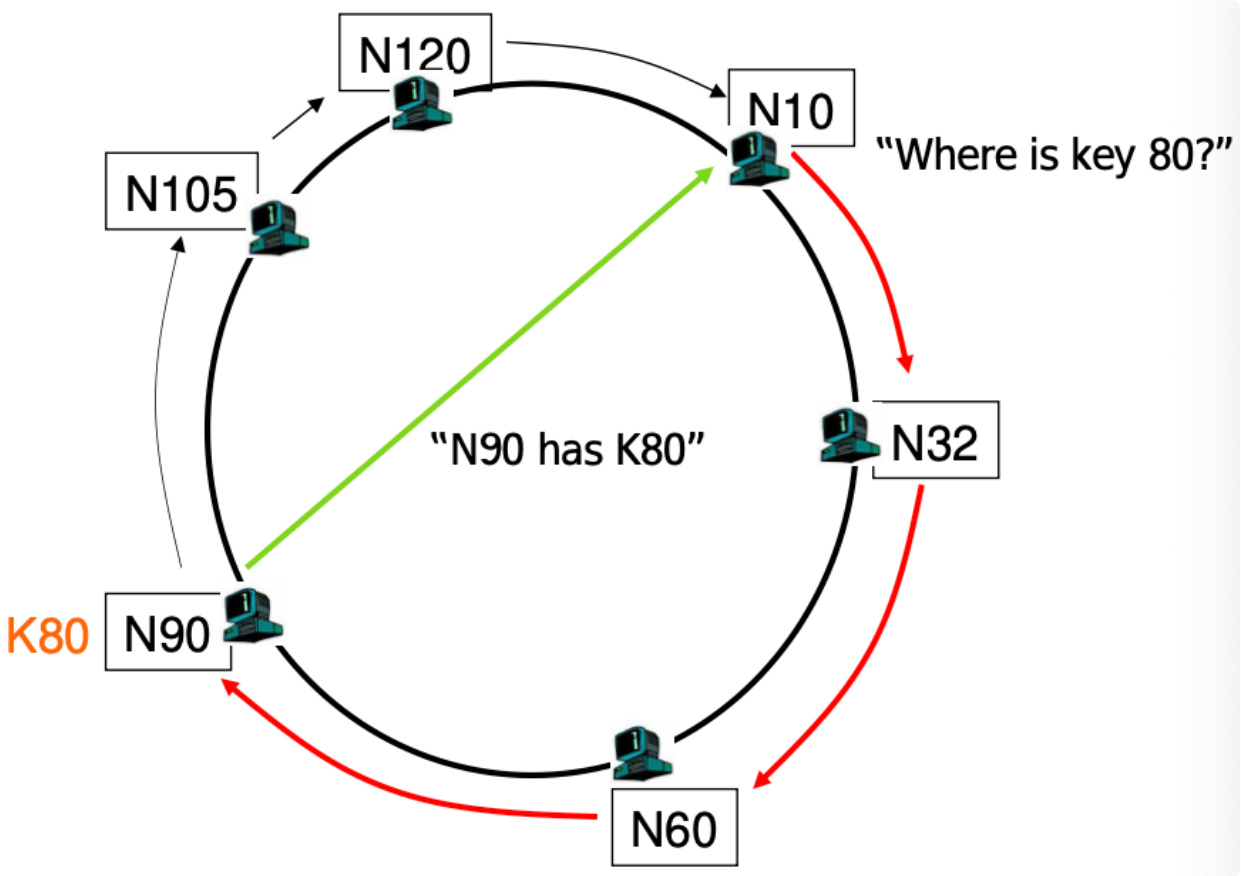
In Chord, an early implementation of Distributed Hash Table, nodes and keys are organised in a **logical ring**:

- Each node is assigned a unique $m$ bit identifier, which is usually the hash of the IP address of that node
- Every item is assigned a unique $m$ bit key, which is usually the hash of the item

The item with key $k$ is **managed** by the node with the smallest id such that $\text{id} \geq k$, the **successor**.

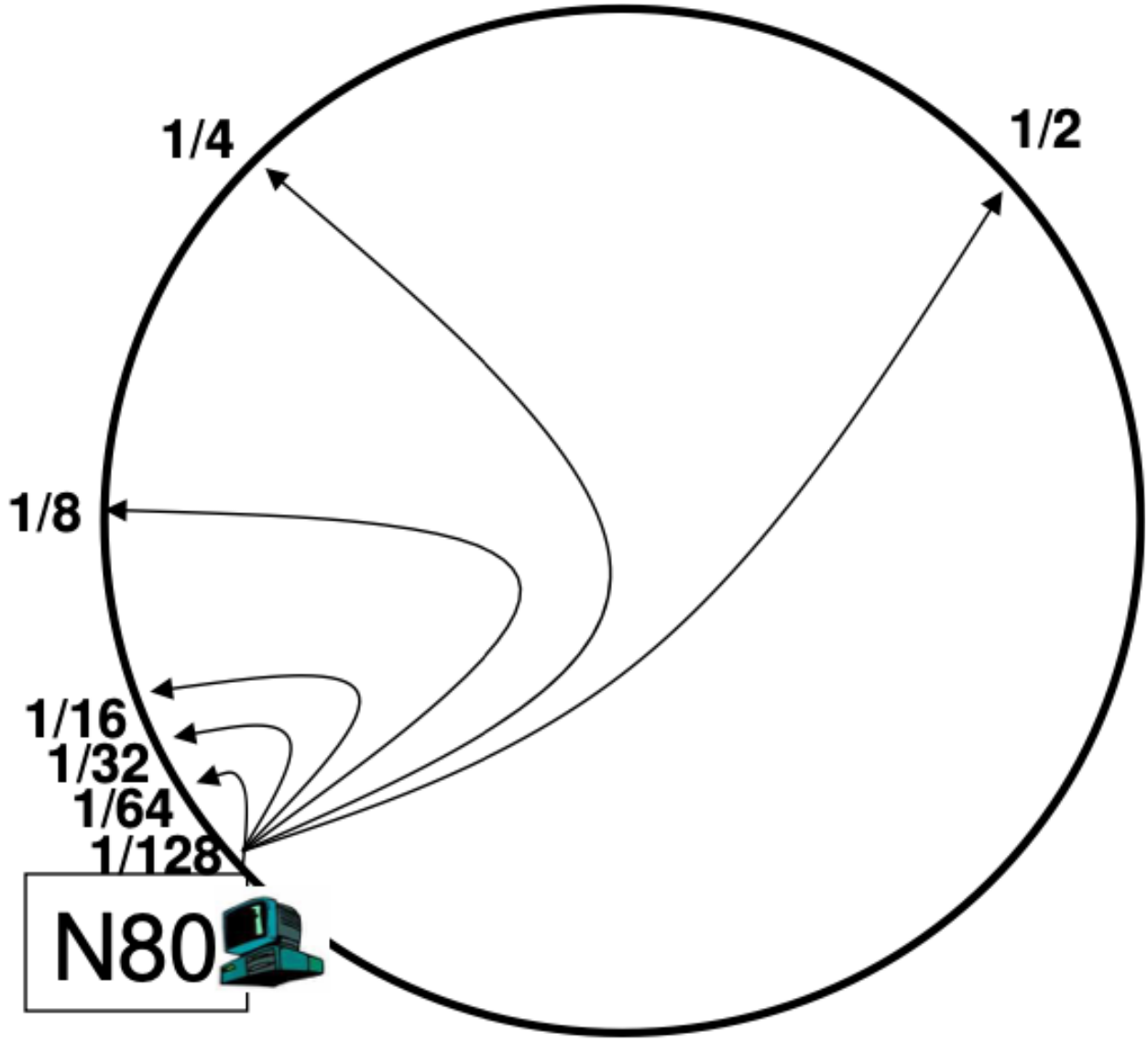



In a basic lookup method, each node keeps track of its successor and search is performed linearly:



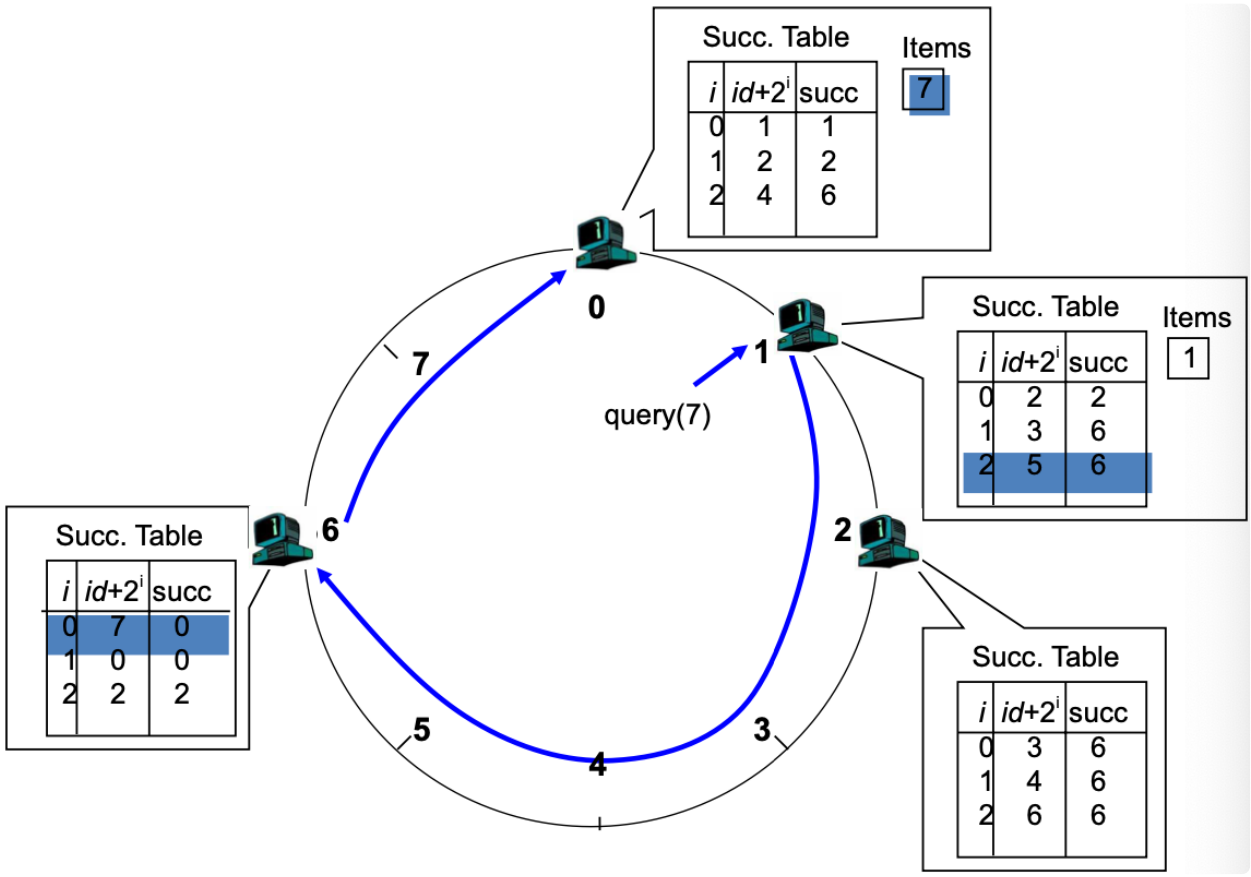

This approach is not very smart. A better implementation of search is the so-called **finger table**: each node maintains a finger table with $m$ entries and entry $i$ in the finger table of node $n$ is the first node whose ID is greater or equal to $n + 2^i$, for $i = 0, \ldots, m - 1$.

In other words, the $i$-th finger points $1/2^{m-i}$ way around the ring:




Upon receiving a query for an item with key $k$, a node:

1. Checks whether it stores the item locally
2. If not, forwards the query to the largest node in the second column of its successor table that does not exceed $k$




In Chord, routing time is $O(\log N)$ hops, where $N$ is the number of nodes in the network.

### Hierarchical approaches

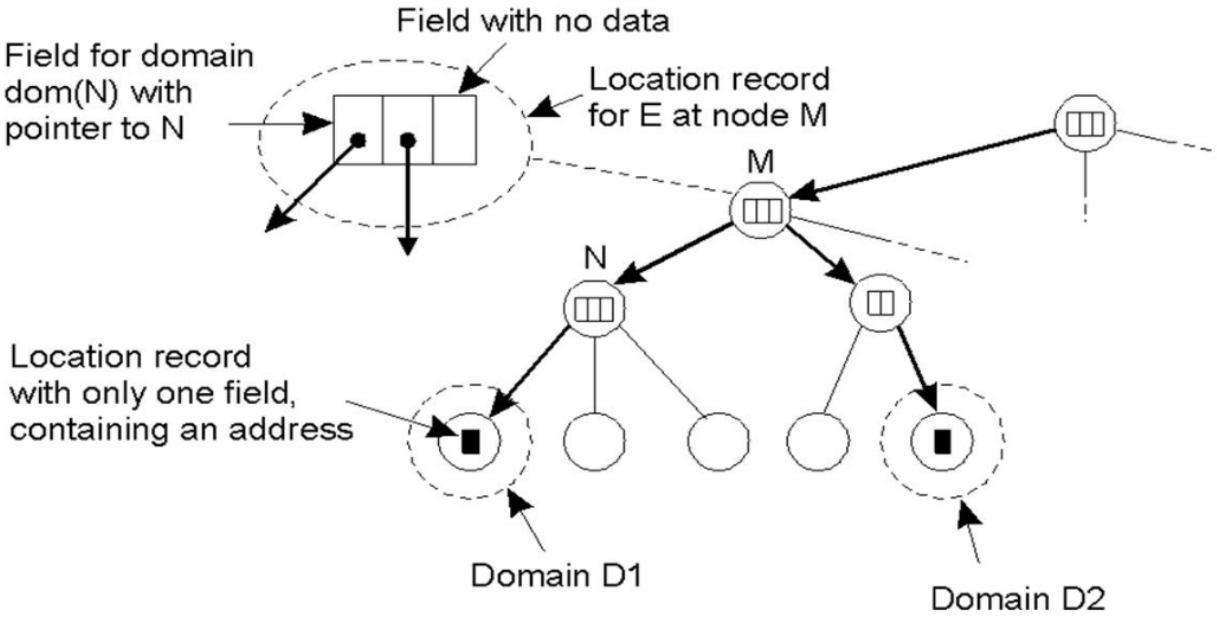
Hierarchical approaches are based on two-tier, home-based approaches: first, a node looks in its local registry to see if the entity is there; otherwise, it contacts the home.

Hierarchical approaches are a generalisation of the two-tier approach described above:

- The network is divided into **domains**
- The **root domain** spans the entire network, while **leaf domains** are typically a LAN or a mobile phone cell
- Each domain has an associated **directory node** that keeps track of the entities in that domain

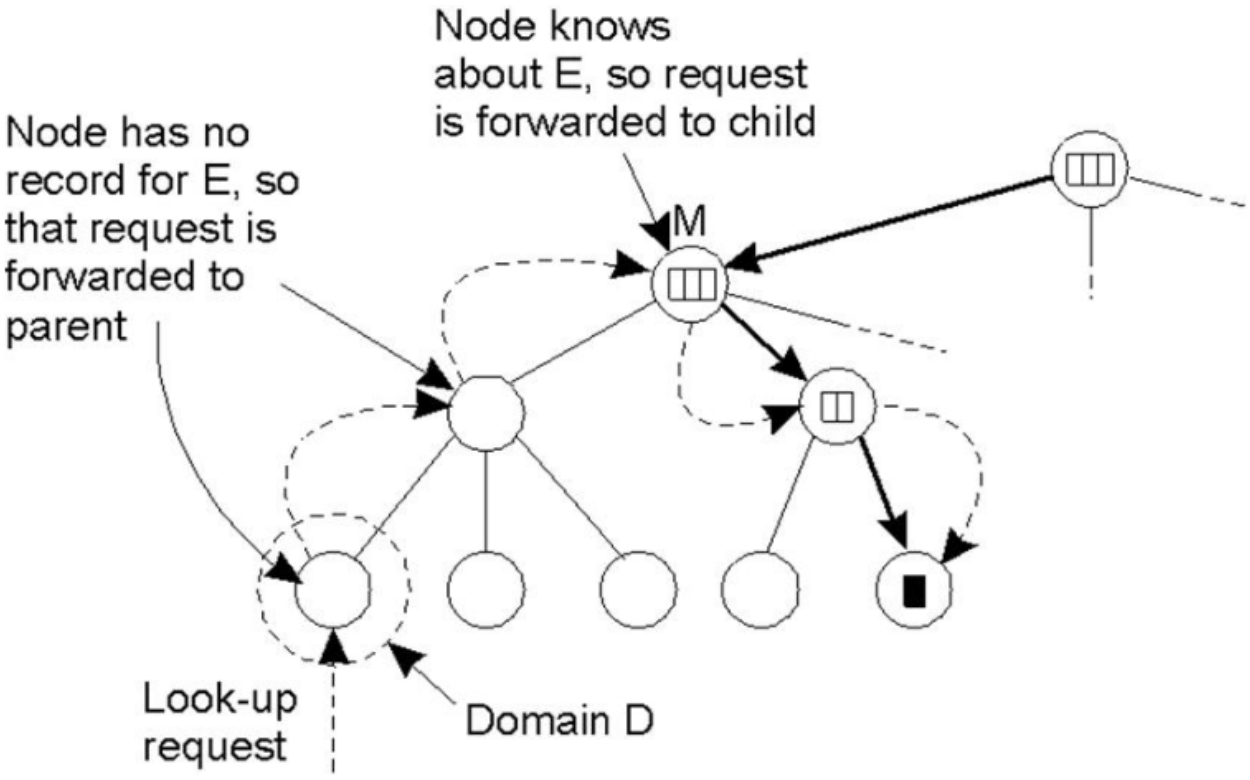
In a hierarchical approach, the root has entries for every entity in the network and entries point to the **next subdomain**. A leaf contains the **address of an entity**.

Entities may have multiple addresses in different leaf domains.



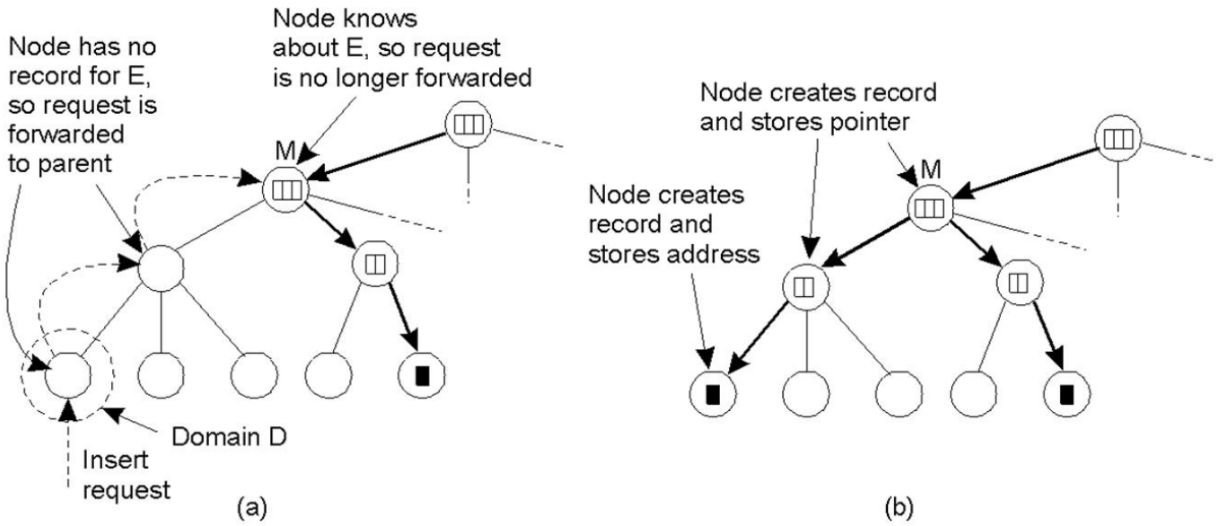

In hierarchical approaches, lookup may start **anywhere**, but the search is always **local-first**. If the element the server is looking for is not in the current node, the lookup request is forwarded to a parent; if an entry is found, the lookup is forwarded to a child until a leaf holding the concrete entry is found.

Through this method, queries achieve **locality**, but the root has information on all entries, which is a risk.



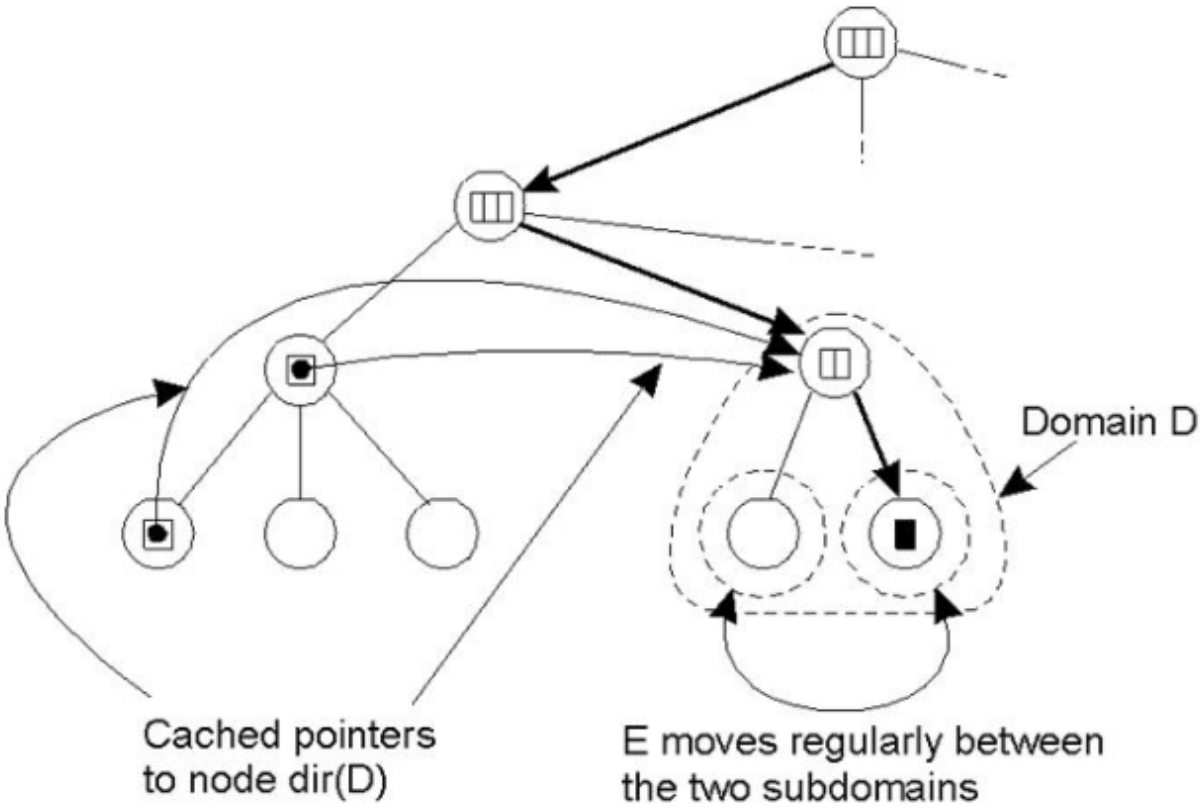

Updates start with an insert request from a new location. Location records are created either top-down or bottom-up. Bottom-up has the advantage of **allowing immediate queries**.

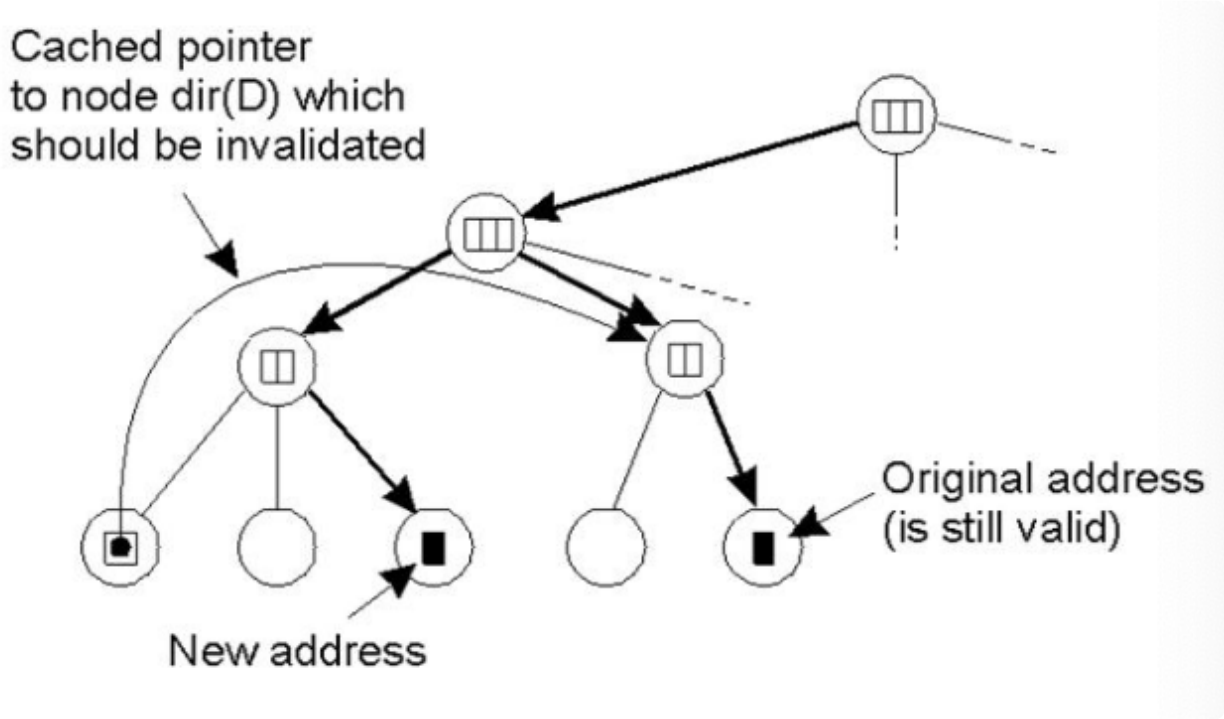
Deleting proceeds from the old node upwards, stopping when a node with multiple children is reached.




## Optimisations and open issues

**Caching** addresses directly is generally inefficient, but it is possible to shortcut a lookup if information about **mobility patterns** is available:






As far as **scalability** is concerned, the root node is expected to hold data for all entities. Records are generally small, but lookups through the root can create a bottleneck.

The root can be distributed, but then the allocation of entities to roots can be difficult.

---

## Structured naming

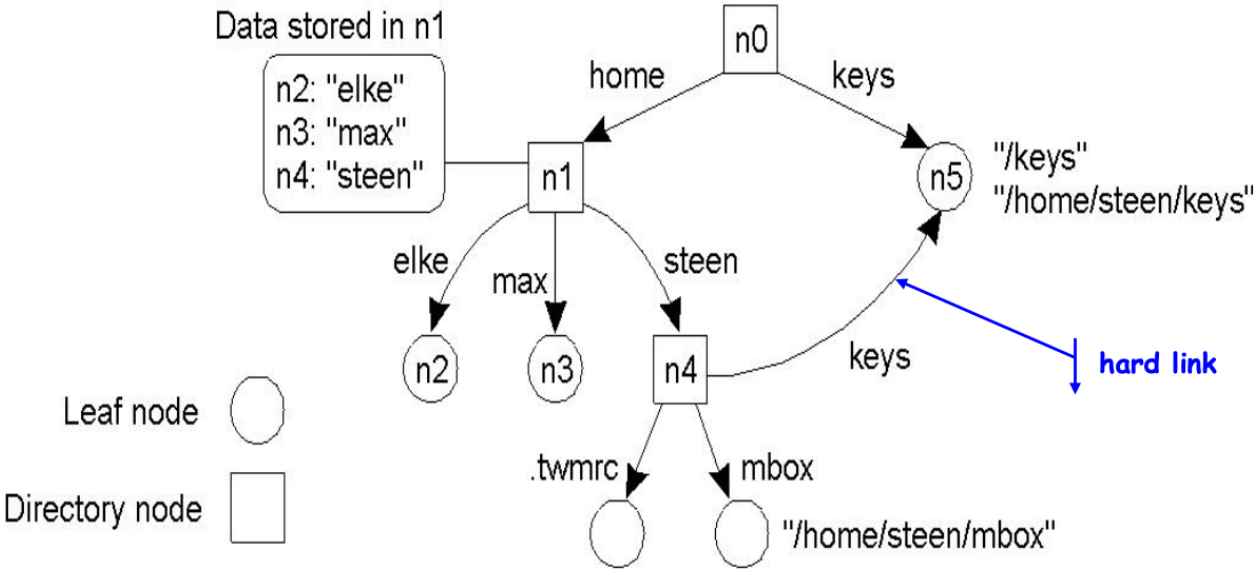
In a structured naming system, names are organised in a **name space**, which is a **labeled graph** composed of **leaf nodes** and **directory nodes**.

A leaf node represents a **named entity**, and it stores information about the entity it refers to. They include, at least, its identifier or address.

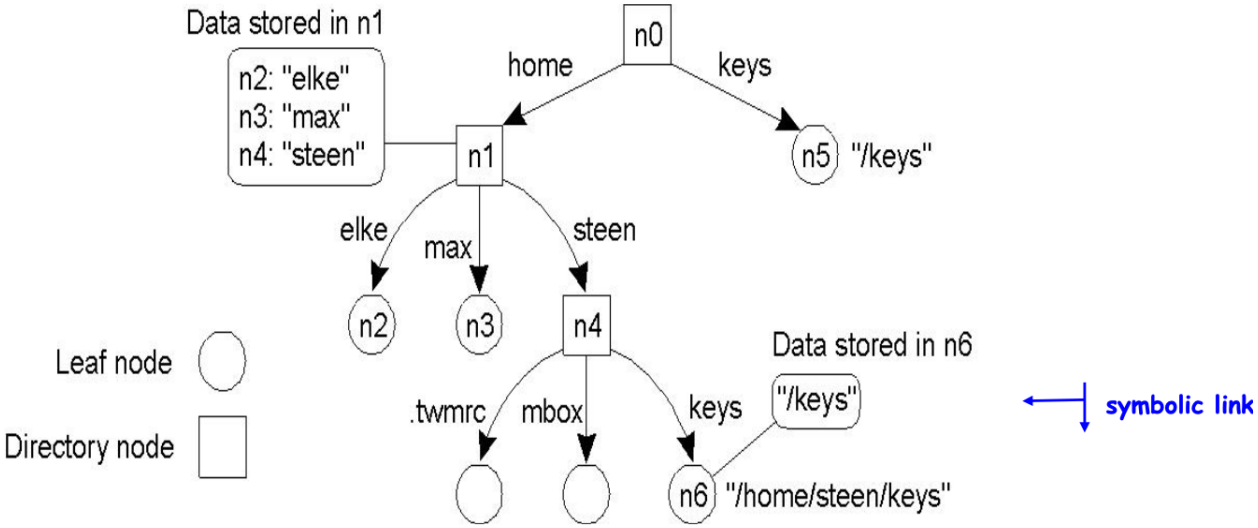
A directory node has a number of **labeled outgoing edges**, each pointing to a different node. A node is a **special case** of an entity with an identifier or address.

In name spaces, resources are referred to through **path names**, which can be **absolute** or **relative**. Multiple path names may refer to the same entity (*hard linking*) or they may store absolute path names of the entity they refer to (*symbolic linking*).

A file system works on structured naming. Here's an example of hard link:




And here's an example of symbolic link:




## Name space distribution

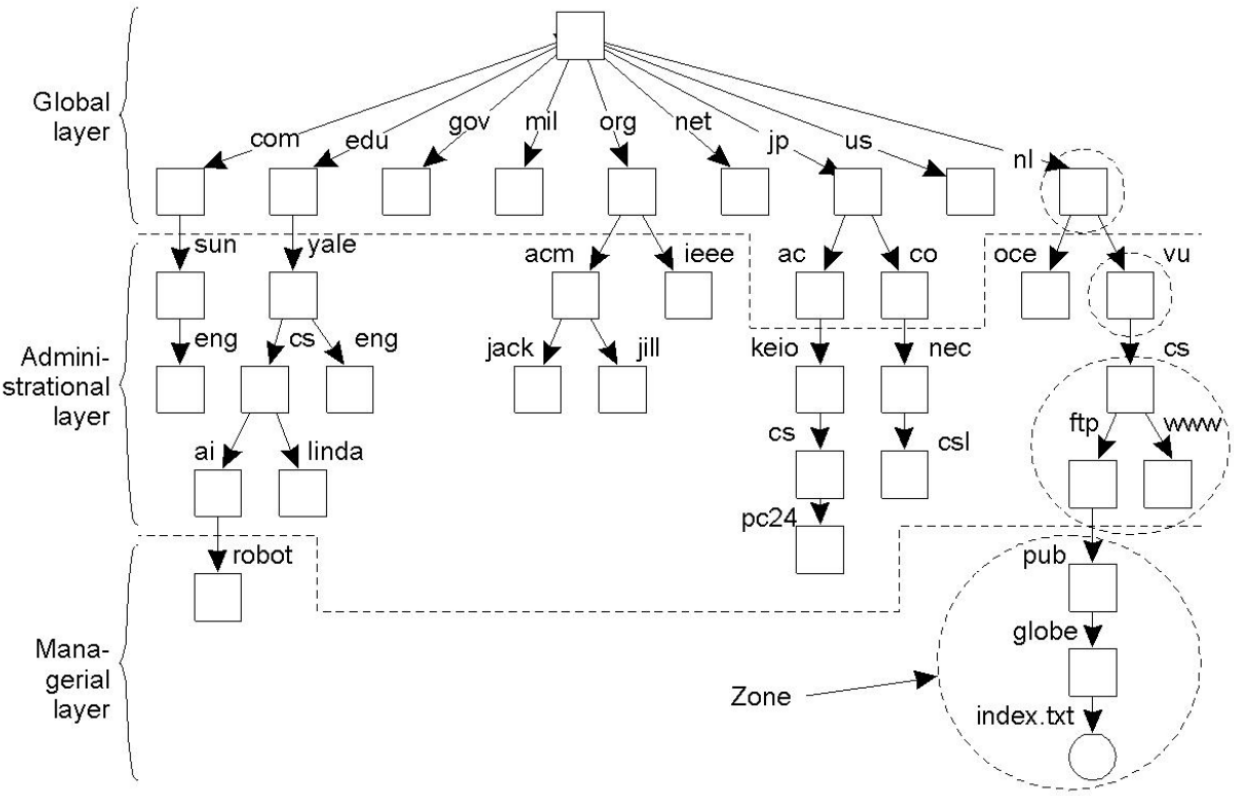
Name spaces for large scale, possibly worldwide, distributed systems are often distributed among different **name servers**, usually organised hierarchically.

The name space is **partitioned** into **layers**. Typically there are three layers:

- **Global level**: consists of the high-level directory nodes. These directory nodes have to be jointly managed by different administrations
- **Administrational level**: contains mid-level directory nodes that can be grouped in such a way that each group can be assigned to a separate administration
- **Managerial level**: consists of low-level directory nodes within a single administration. The main issue with this layer is effectively mapping directory nodes to local name servers

Each node of the name space is assigned to a name server.

An example of distributed name space is the **DNS**:



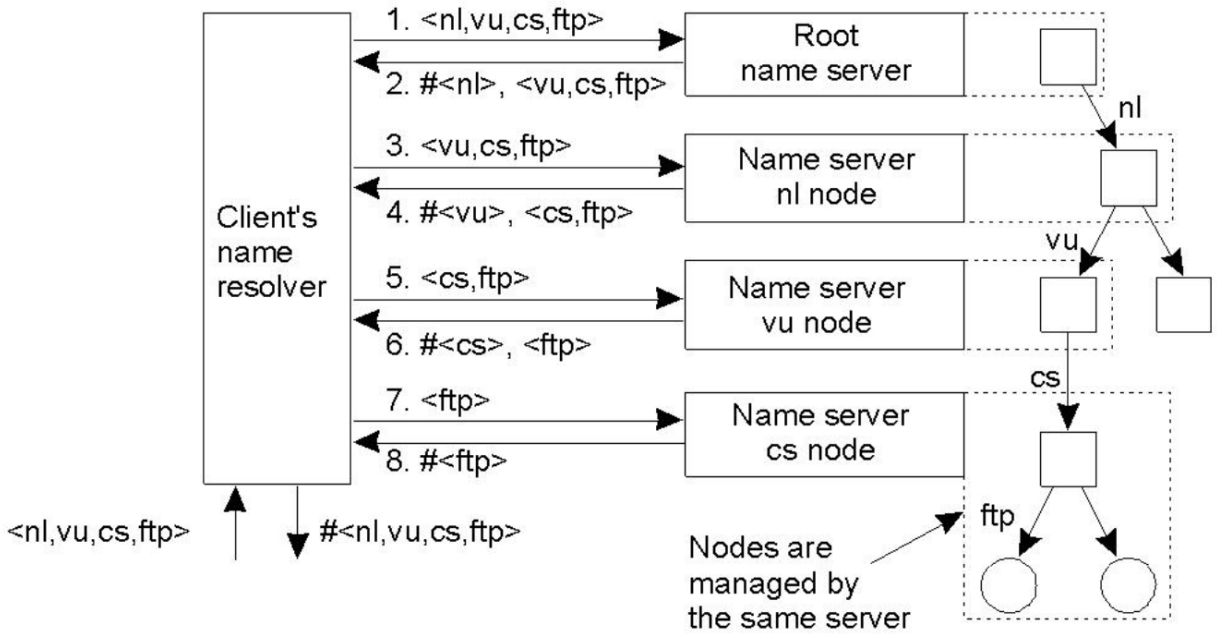

Let's compare the different layers a name space can be partitioned in:

| Characteristic | Global | Administrational | Managerial |
|---|---|---|---|
| Geographic scale of network | Worldwide | Organisation | Department |
| Total number of nodes | Few | Many | Vast number |
| Responsiveness to lookups | Seconds | Milliseconds | Immediate |
| Update propagation | Lazy | Immediate | Immediate |
| Number of replicas | Many | Few to none | None |
| Is client-side caching applied? | Yes | Yes | Sometimes |

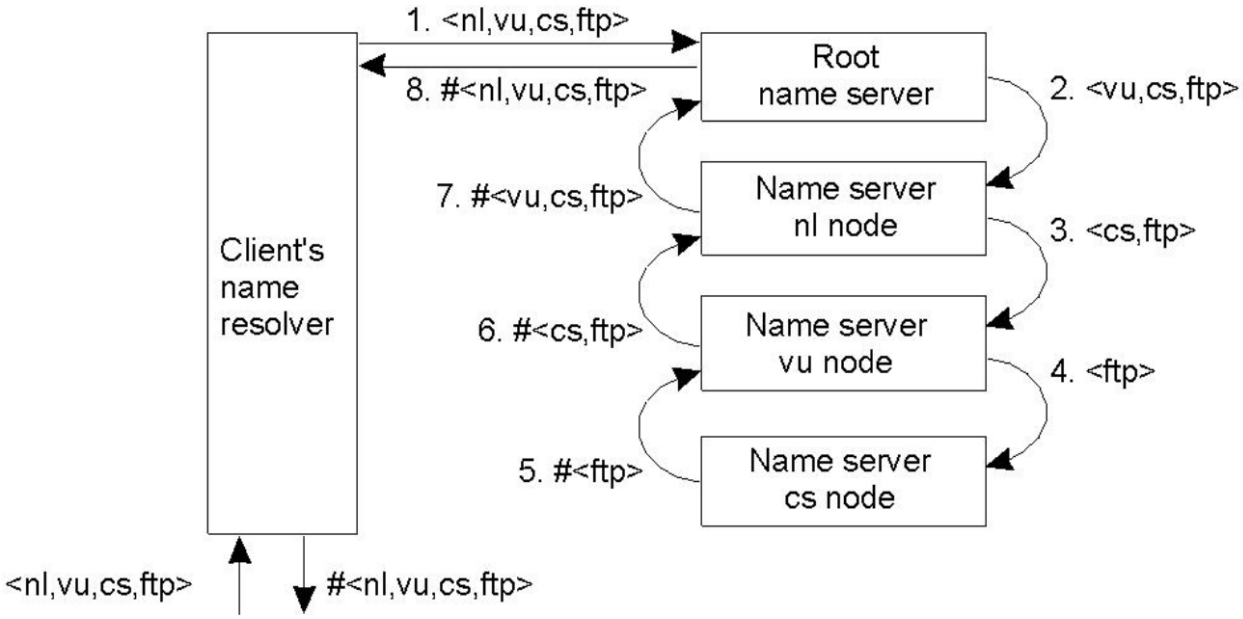
## Name resolution

Name resolution can be performed in two main ways: **iteratively** and **recursively**.

Here's an example of iterative name resolution:




Here's an example of recursive name resolution:




Recursive resolution renders communication less expensive and caching more effective, since it can be performed along the resolution chain, leading to faster lookups. Recursive resolution leads to higher demand on each name server though.

### DNS structure in practice

DNS uses a hierarchically-organised name space as a root tree with no hard links. Each subtree is named **domain** and belongs to a **separate authority**, while each name server is responsible for a zone.

Names are case-insensitive, up to 255 characters, with a maximum of 63 characters per label.

Each node may represent several entities through a set of **resource records**:

| Type of record | Associated entity | Description |
|---|---|---|
| SOA | Zone | Holds information on the represented zone |
| A | Host | Contains an IP address of the host represented by this node |
| MX | Domain | Refers (symlink) to a mail server to handle mail addressed to this node |
| SRV | Domain | Refers (symlink) to a server handling a specific service, like HTTP |
| NS | Zone | Refers (symlink) to a name server that implements the represented zone |
| CNAME | Node | Symbolic link with the primary name of the represented node |
| PTR | Host | Contains the canonical name of a host, for reverse lookups |
| HINFO | Host | Holds information on the host represented by this node |
| TXT | Any kind | Contains any entity-specific information considered useful |

### DNS resolution in practice

When executing a DNS query, clients can request the resolution mode, but servers are not obliged to comply.

In practice, a mix of iterative and recursive resolution is used, although global name servers typically support iterative resolution only.

Global servers are mirrored, and IP anycast is used to route queries among them.

Caching and replication are massively used: secondary DNS server are periodically brought up-to-date by the primary ones and a TTL (*Time to Live*) attribute is associated to information, determining its persistence in the cache.

Transient inconsistencies are allowed, and only host information is stored, but, in principle, other information could be stored.

### DNS and mobile entities

DNS works well based on the assumptions that:

- Content of global or administrational layers is quite stable
- Content of managerial layer changes often, but requests are served by name servers in the same zone, therefore updates are efficient

If a host wants to "move", there are two possibilities:

- If it stays within the original domain, only the database of the domain name servers need to be updated
- If it moves to an entirely different domain, then there are some approaches to solve this problem:
  - The DNS servers of the old domain could provide the IP address of the new location
  - The DNS servers of the old domain could provide the name of the new location (basically a symlink)

It is still better to keep the name of the domain that moves, since applications and users rely on it.

---

## Attribute based naming

As more information is made available, it becomes important to effectively search for items. A solution to this problem is to refer to entities not with their name, but with a **set of attributes**, which code their properties.

In attribute based naming, each entity has a set of associated attributes and the name system can be queried by searching for entities given the values of their attributes.

Attributed based naming systems are usually called **directory services** and are usually implemented using **DBMS technology**.

### Structured, attribute based naming: LDAP

A common approach to implement a distributed directory service is to combine structured with attribute based naming.

The **LDAP** (*Lightweight Directory Access Protocol*) directory service is becoming the de-facto standard in this field. It is derived from the OSI X.500 directory service.

An LDAP directory consists of a number of record called **directory entries**:

- Each one is made as a collection of attribute-value pairs
- Each attribute has a type
- Both single-valued and multiple-valued attributes exist

Some attributes are part of the standard.

The collection of all records in a LDAP directory service is called **Directory Information Base** or **DIB**.

In a DIB, each record has a unique name defined as a sequence of naming attributes, also known as **Relative Distinguished Name** or **RDN**. This approach leads to build a **Directory Information Tree** or **DIT**. A node in a LDAP naming graph can thus simultaneously represent a directory in a traditional sense.

When dealing with large scale directories, the DIB is usually partitioned according to the DIT structure, as is the case for DNS. Each server is then known as **Directory Service Agent** or **DSA**, while the client is known as **Directory User Agent** or **DUA**.

What LDAP adds to a standard hierarchical naming schema is its searching ability.

Microsoft's Active Directory is based on LDAP together with several other technologies.

Another directory service that's becoming a standard has been developed in the context of grid computing and web services and is known as **Universal Directory and Discovery Integration**, or **UDDI**.

---

## Removing unreferenced entities

Naming provides a global referencing service and so entities accessed through **stale bindings** should be **removed**.

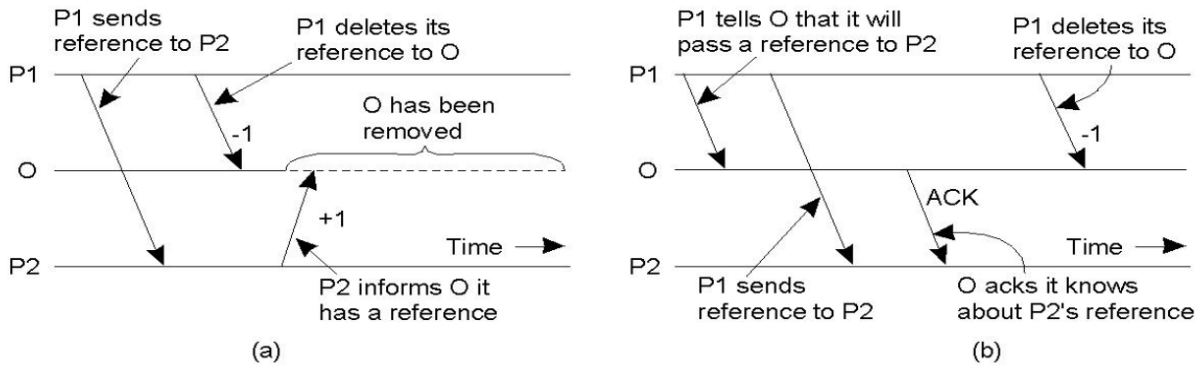
Automatic **garbage collection** is common in conventional systems.

Distribution greatly complicates matters, due to the lack of global knowledge about who's using what and to network failures.

## Reference counting

One way to remove unreferenced entities is through reference counting, where each object keeps track of how many other objects it is referenced by. If the counter goes to zero, the object can be deallocated from memory.

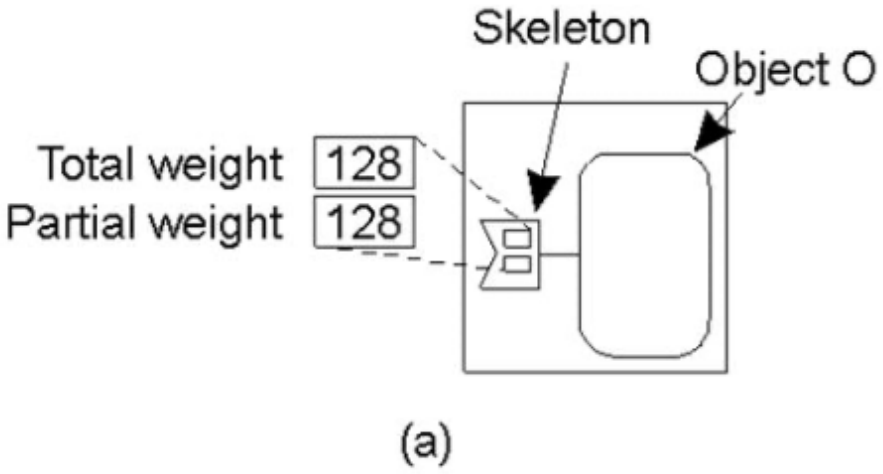
Reliability in a system of this kind must be ensured, typically through acknowledgements and duplicate detection and elimination. Passing a reference requires three messages, which leads to potential performance issues in large-scale systems.



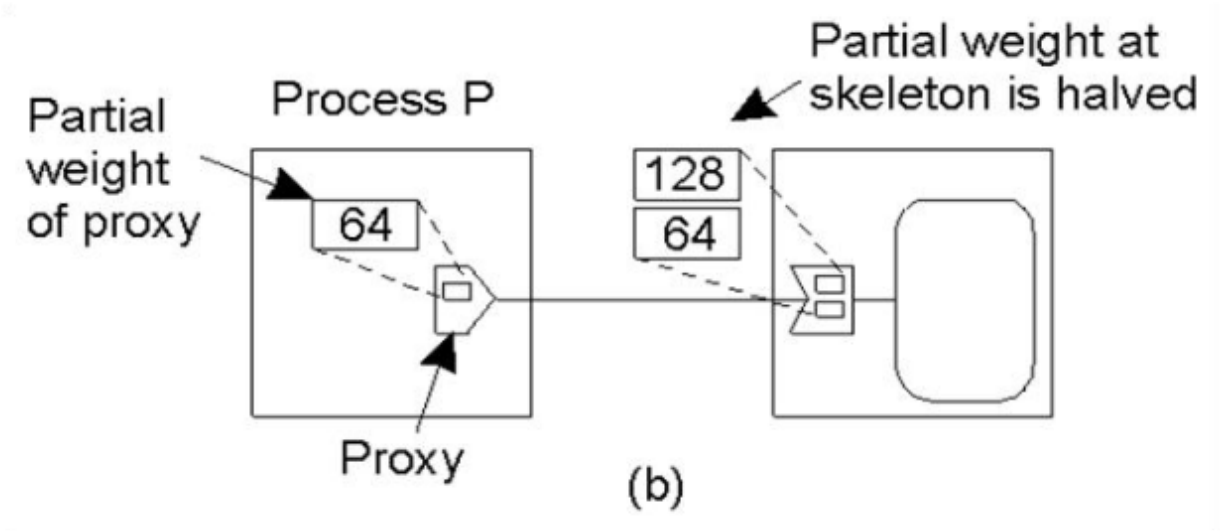

## Weighted reference counting

Weighted reference counting tries to circumvent the problem found in normal reference counting by communicating only counter **decrements**. This requires an additional counter.

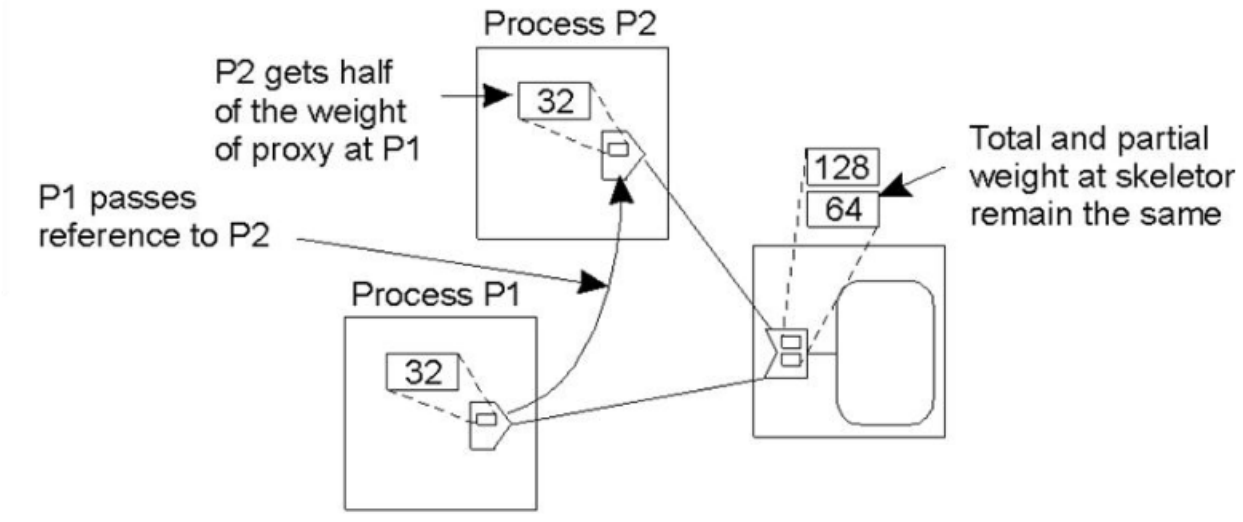
Removing a reference subtracts the proxy partial counter from the total counter of the skeleton: when the total and partial weights become equal, the object can be removed.




(a)




(b)

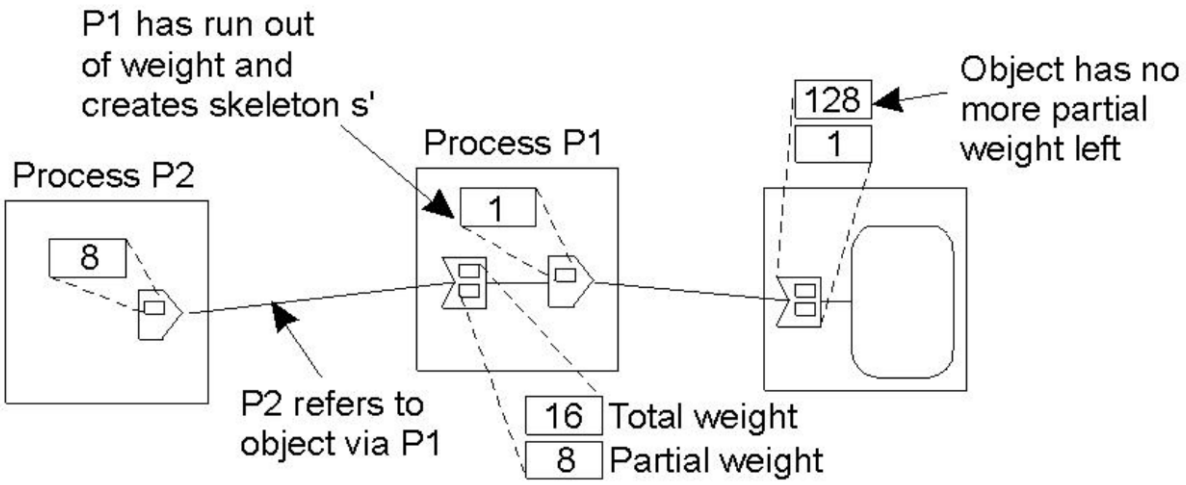



(c)

This poses another problem: only a fixed number of references can be created.

### Weighted reference counting using indirection

This method removes the limitation on the overall number of references, but introduces an additional hop to access the target object.




### Reference listing

Instead of keeping track of the number of references, we can keep track of the **identities** of the proxies. This brings forth two advantages:

- Insertion and deletion of a proxy is **idempotent**: requests for insertion and deletion can be issued multiple times with the same effect; non-reliable communication can then be used.
- It's easier to maintain the list consistent with respect to network failures

This method still suffers from race conditions when copying references, though.

This is the method that is used by Java RMI.

### Identifying unreachable entities

**How to detect entities that are disconnected from the root set?**

Tracing-based garbage collection techniques are generally used in small systems, since they require knowledge about all entities and have therefore poor scalability.

On centralised systems, **mark and sweep** is used:

1. Mark accessible entities by following references
    - Initially, all nodes are white

    - A node is coloured grey when reachable from a root but some of its references still need to be evaluated
    - A node is coloured black after it turned grey and all its outgoing references have been marked grey
2. Examine memory to locate entities that were marked white in the previous step in order to delete them

In distributed systems, a similar method is used, called **Emerald distributed object system**: a garbage collector runs on each site and the marking is done as follows:

- An object in process $P$ reachable from a root in $P$ is marked grey together with all the proxies in it
- When a proxy $Q$ is marked grey, a message is sent to its associated skeleton
- An acknowledgement is expected before turning $Q$ black

The sweep process is similar to the one for centralised systems: all white objects are collected locally and the process can start as soon as all local objects are either black or white.

This technique still requires the reachability graph to remain stable, which means that this type of operation blocks the entire system. This leads to scalability issues.